Introduction to I2C and SPI both in-kernel and in user space.

Michael Welling
Founder
QWERTY Embedded Design, LLC

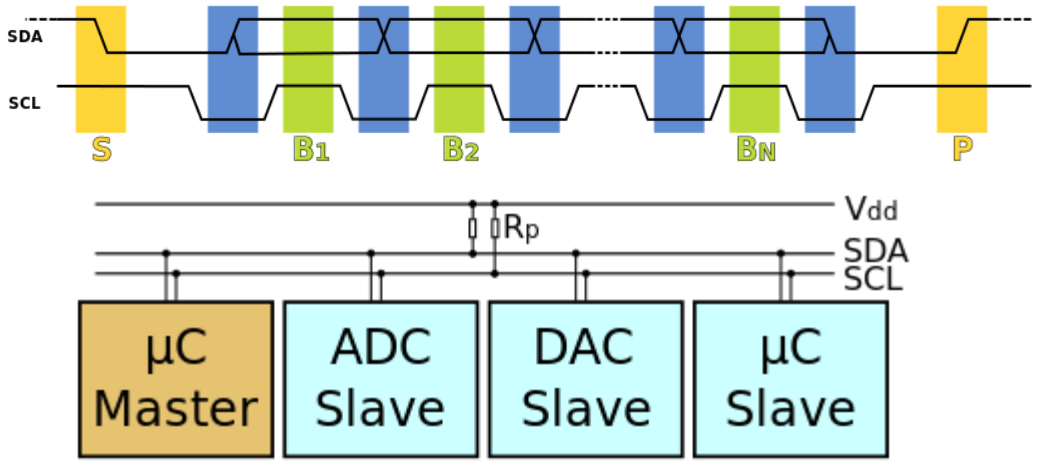
#lfelc @QwertyEmbedded

I2C Overview

- What is I2C?
- Example I2C Devices
- I2C Protocol
- Linux I2C Subsystem
- Linux I2C Drivers
  - I2C Bus Drivers
  - I2C Device Drivers
  - I2C Slave Interface
- Instantiating I2C Devices
- User space tools
- Demo

What is I2C?

- I^2C stands for inter-integrated circuit
- First developed by Philips Semiconductor in 1982, currently owned by NXP Semiconductors
- Synchronous multi-master multi-slave serial bus
- Half duplex protocol
- Open-drain signaling
- Only two signal wires
  - SDA: serial data
  - SCL: serial clock



en:user:Cburnett, I2C, CC BY-SA 3.0

https://en.wikipedia.org/wiki/I%C2%B2C

What is I2C?

- I^2C addressing is typically 7 bits per the original specification
- I^2C clock speed is typically 100KHz per the original specification
- Later versions of the specification introduced faster clock modes and 10 bit addressing
  - Version 1 added 400KHz Fast mode and 10 bit addressing
  - Version 2 added 3.4MHz Hs mode
  - Version 3 added 1MHz Fast mode+ and ID mechanism
  - Version 4 added unidirectional 5Mhz Ultra Fast mode

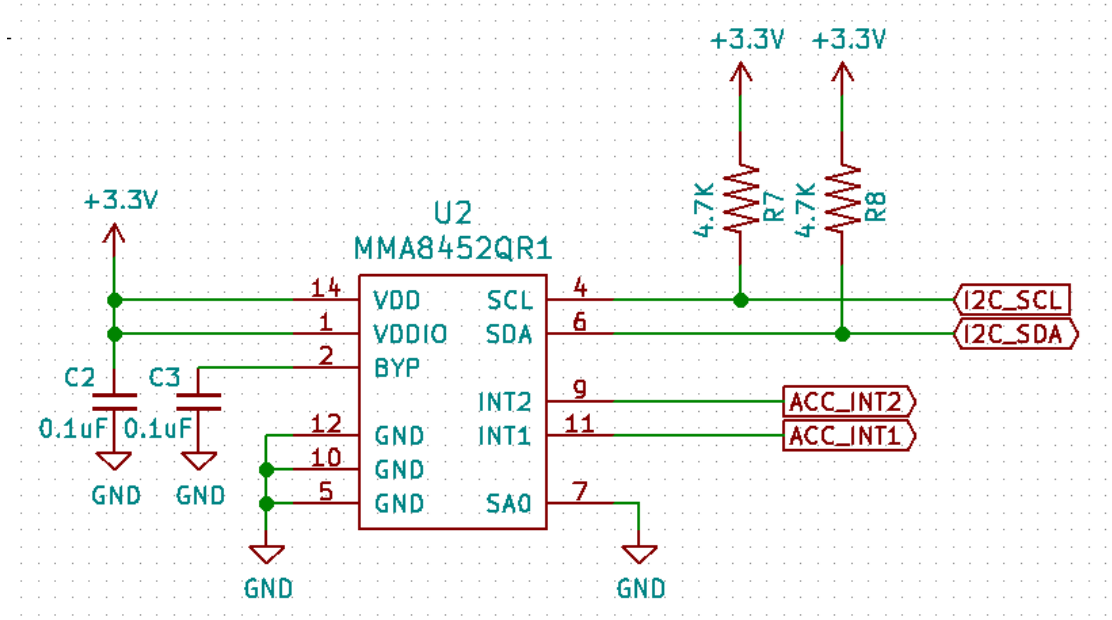
What is I2C?

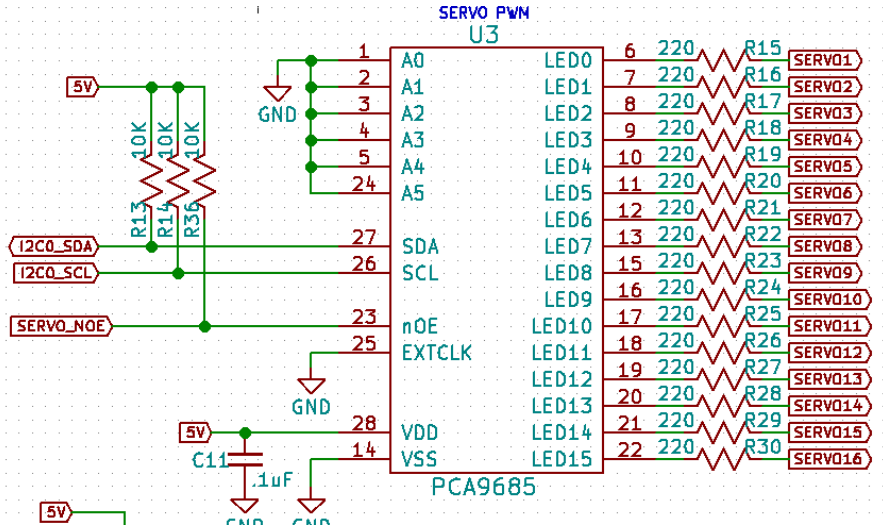
- SMBus (System Management Bus) is a subset of I^2C defined by Intel
  - Typically used on PC motherboards for power control and sensors
  - Stricter tolerances on voltage levels and timing
  - Adds an optional software level address resolution protocol

Example I2C Devices

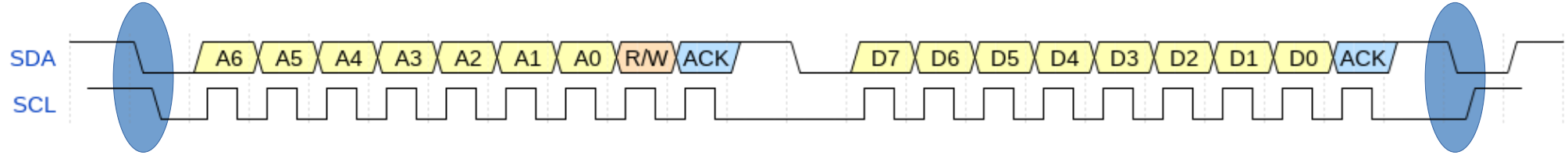
- Real time clock
- EEPROM
- Analog converters (ADC, DAC)
- Sensors (Temperature, Pressure)
- Microcontrollers
- Touchscreen controllers
- GPIO expanders
- Monitor and TV adapters

Example I2C Hardware





I2C Protocol



Start

SDA goes low before SCL to signal the start of transmission.

Addr

7 bit address that determines the slave device to be accessed.

R/W

Transaction data direction bit. (1 = read, 0 = write)

Data

Byte data read from or written to the slave device. Can be multiple bytes.

ACK

Acknowledge bit. (0 = ack, 1 = nak)

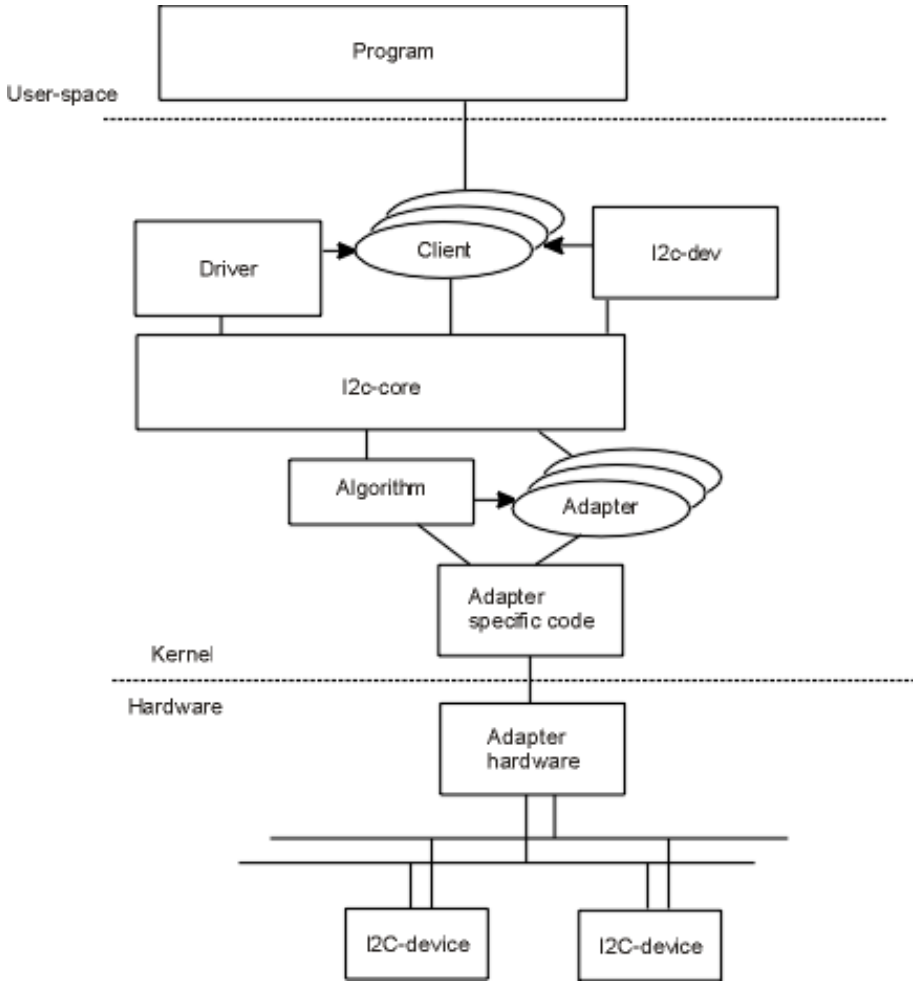
Stop

SDA goes high after SCL to signal the end of transmission.

Linux I2C Subsystem

- Early implementations were from Gerd Knorr and Simon G. Vogl.
- Migrated to the device model by Greg KH in late 2.5 versions of Linux.
- Integrated into standard device driver model by David Brownell and Jean Delvare in Linux 2.6.
- Currently maintained by Wolfram Sang.

Linux I2C Subsystem



https://i2c.wiki.kernel.org

Linux I2C Subsystem

List: linux-i2c; (subscribe / unsubscribe)

Info:

```
Linux kernel I2C bus layer mailing list.
Archives:

http://marc.info/?l=linux-i2c
http://www.spinics.net/lists/linux-i2c/
```

Linux I2C Drivers

I2C Bus Drivers

Bus → Algorithm

Adapter

An Algorithm driver contains general code that can be used for a whole class of I2C adapters. Each specific adapter driver either depends on one algorithm driver, or includes its own implementation.

https://www.kernel.org/doc/Documentation/i2c/summary

I2C Bus Driver

- Define and allocate a private data struct (contains **struct i2c_adapter**)
- Fill algorithm struct
  - **.master_xfer()** – function to perform transfer
  - **.functionality()** – function to retrieve bus functionality.
- Fill adaptor struct
  - **i2c_set_adapdata()**
  - **.algo** – pointer to algorithm struct
  - **.algo_data** – pointer the private data struct
- Add adapter
  - **i2c_add_adapter()**

Linux I2C Drivers

I2C Device Drivers

Device → Driver

Client

A Driver driver (yes, this sounds ridiculous, sorry) contains the general code to access some type of device. Each detected device gets its own data in the Client structure. Usually, Driver and Client are more closely integrated than Algorithm and Adapter.

https://www.kernel.org/doc/Documentation/i2c/summary

Linux I2C Drivers

I2C Device Driver
i2c_driver

```
static struct i2c_driver foo_driver = {
        .driver = {
                .name   = "foobar",
                .of_match_table = of_match_ptr(foo_dt_ids)
        },
        .id_table       = foo_idtable,
        .probe          = foo_probe,
        .remove         = foo_remove,
};
```

I2C Device Driver
i2c_device_id / of_device_id

```
static struct i2c_device_id foo_idtable[] = {
        { "foo", 0 },
        { }
};

MODULE_DEVICE_TABLE(i2c, foo_idtable);

static const struct of_device_id foo_dt_ids[] = {
        { .compatible = "foo,bar", .data = (void *) 0xDEADBEEF },
        { }
};
MODULE_DEVICE_TABLE(of, foo_dt_ids);
```

I2C Device Driver
Probe function

```
static int foo_probe(struct i2c_client *client, const struct i2c_device_id *id)
{
	int ret;

	pr_info("foo_probe called\n");

	if (client->dev.of_node) {
		pr_info("device tree instantiated probe. data = %x\n",
			(unsigned int)of_device_get_match_data(&client->dev));
	}

	ret = i2c_smbus_read_byte_data(client, 0x0d);
	pr_info("i2c read byte = %x\n", ret);
	if (ret < 0)
		return ret;

	return 0;
}
```

I2C Device Driver
Remove function

```
static int foo_remove(struct i2c_client *client)
{
        /* do any cleanup here*/

        pr_info("foo_remove called\n");
        return 0;
}
```

Each client structure has a special data field that can point to any structure at all. You should use this to keep device-specific data.

I2C Device Driver
Client data

Each client structure has a special data field that can point to any structure at all. You should use this to keep device-specific data.

```
/* store the value */
void i2c_set_clientdata(struct i2c_client *client, void *data);

/* retrieve the value */
void *i2c_get_clientdata(const struct i2c_client *client);
```

I2C Device Driver
Initializing the driver

```
static int __init foo_init(void)
{
	return i2c_add_driver(&foo_driver);
}
module_init(foo_init);

static void __exit foo_cleanup(void)
{
	i2c_del_driver(&foo_driver);
}
module_exit(foo_cleanup);
```

The module_i2c_driver() macro can be used to reduce above code.

```
module_i2c_driver(foo_driver);
```

I2C Device Driver
Plain I2C API

int **i2c_master_send**(struct i2c_client *client, const char *buf,
int count);

int **i2c_master_recv**(struct i2c_client *client, char *buf, int count);

These routines read and write some bytes from/to a client. The client contains the I2C address, so you do not have to include it. The second parameter contains the bytes to read/write, the third the number of bytes to read/write (must be less than the length of the buffer, also should be less than 64k since msg.len is u16.) Returned is the actual number of bytes read/written.

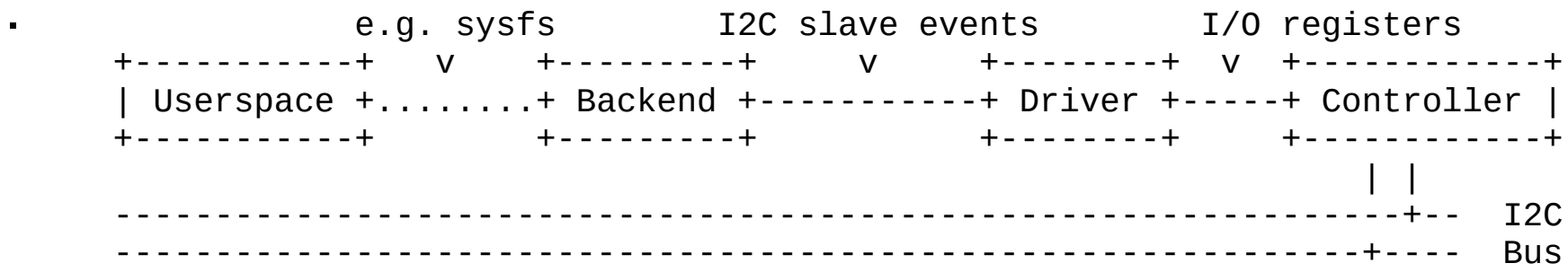
int **i2c_transfer**(struct i2c_adapter *adap, struct i2c_msg *msg,
int num);

This sends a series of messages. Each message can be a read or write, and they can be mixed in any way. The transactions are combined: no stop condition is issued between transaction. The i2c_msg structure contains for each message the client address, the number of bytes of the message and the message data itself.

I2C Device Driver
SMBus I2C API

```
s32 i2c_smbus_read_byte(struct i2c_client *client);
s32 i2c_smbus_write_byte(struct i2c_client *client, u8 value);
s32 i2c_smbus_read_byte_data(struct i2c_client *client, u8 command);
s32 i2c_smbus_write_byte_data(struct i2c_client *client,
                              u8 command, u8 value);
s32 i2c_smbus_read_word_data(struct i2c_client *client, u8 command);
s32 i2c_smbus_write_word_data(struct i2c_client *client,
                              u8 command, u16 value);
s32 i2c_smbus_read_block_data(struct i2c_client *client,
                              u8 command, u8 *values);
s32 i2c_smbus_write_block_data(struct i2c_client *client,
                               u8 command, u8 length, const u8 *values);
s32 i2c_smbus_read_i2c_block_data(struct i2c_client *client,
                                  u8 command, u8 length, u8 *values);
s32 i2c_smbus_write_i2c_block_data(struct i2c_client *client,
                                   u8 command, u8 length,
                                   const u8 *values);
```

I2C Slave Interface




```
echo slave-24c02 0x1064 > /sys/bus/i2c/devices/i2c-1/new_device
```

https://www.kernel.org/doc/Documentation/i2c/slave-interface

Instantiating I2C Devices

Device tree

```
i2c1: i2c@400a0000 {
        /* ... master properties skipped ... */
        clock-frequency = <100000>;
        flash@50 {
                compatible = "atmel,24c256";
                reg = <0x50>;
        };
        pca9532: gpio@60 {
                compatible = "nxp,pca9532";
                gpio-controller;
                #gpio-cells = <2>;
                reg = <0x60>;
        };
};
```

Platform device

```
static struct i2c_board_info h4_i2c_board_info[] __initdata = {
        {
                I2C_BOARD_INFO("isp1301_omap", 0x2d),
                .irq            = OMAP_GPIO_IRQ(125),
        },
        {       /* EEPROM on mainboard */
                I2C_BOARD_INFO("24c01", 0x52),
                .platform_data  = &m24c01,
        },
        {       /* EEPROM on cpu card */
                I2C_BOARD_INFO("24c01", 0x57),
                .platform_data  = &m24c01,
        },
};
static void __init omap_h4_init(void)
{
        (...)
        i2c_register_board_info(1, h4_i2c_board_info,
                        ARRAY_SIZE(h4_i2c_board_info));
        (...)
}
```

Instantiating I2C Devices

From user space

```
echo eeprom 0x50 > /sys/bus/i2c/devices/i2c-3/new_device
```

https://www.kernel.org/doc/Documentation/i2c/instantiating-devices

User space Tools

- Simple character device driver (i2c-dev)
  - Device nodes at **/dev/i2c-x**
  - Slave address set by **I2C_SLAVE** ioctl.
  - Simple access using **read()** / **write()**
  - **i2c_smbus_{read,write}_{byte,word}_data()**
- i2ctools
  - i2cdetect
  - i2cget
  - I2cset

https://www.kernel.org/doc/Documentation/i2c/dev-interface

User space Tools

From the Linux user space, you can access the I2C bus from the **/dev/i2c-*** device files.

```
debian@beaglebone:~$ ls -l /dev/i2c-*
crw-rw---- 1 root i2c 89, 0 Oct 7 16:40 /dev/i2c-0
crw-rw---- 1 root i2c 89, 1 Oct 7 16:40 /dev/i2c-1
crw-rw---- 1 root i2c 89, 2 Oct 7 16:40 /dev/i2c-2
```

User space Tools

List I2C devices on the bus

```
debian@beaglebone:~$ i2cdetect -y -r 2
0 1 2 3 4 5 6 7 8 9 a b c d e f
00: -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- 1c -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

User space Tools

Dump the register contents of MMA8453

```
debian@beaglebone:~$ i2cdump -y -r 0x00-0x31 2 0x1c
No size specified (using byte-data access)
0 1 2 3 4 5 6 7 8 9 a b c d e f 0123456789abcdef
00: ff fe 00 01 80 41 80 00 00 00 00 01 00 3a 00 00 .?.??A?....?.:..
10: 00 80 00 44 84 00 00 00 00 00 00 00 00 00 00 00 .?.D?...........
20: 00 00 00 00 00 00 00 00 00 00 01 00 00 00 00 00 ..........?.....
30: 00 00
```

User space Tools

Read and write single registers of the MMA8453

```
debian@beaglebone:~$ i2cget -y 2 0x1c 0x0d
0x3a
debian@beaglebone:~$ i2cget -y 2 0x1c 0x2a
0x00
debian@beaglebone:~$ i2cset -y 2 0x1c 0x2a 0x01
debian@beaglebone:~$ i2cget -y 2 0x1c 0x2a
0x01
```

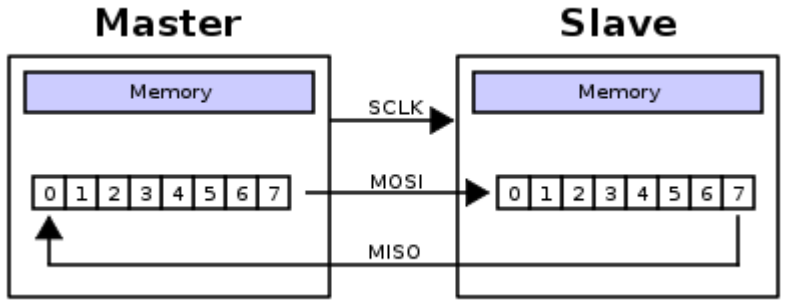
Demo

SPI Overview

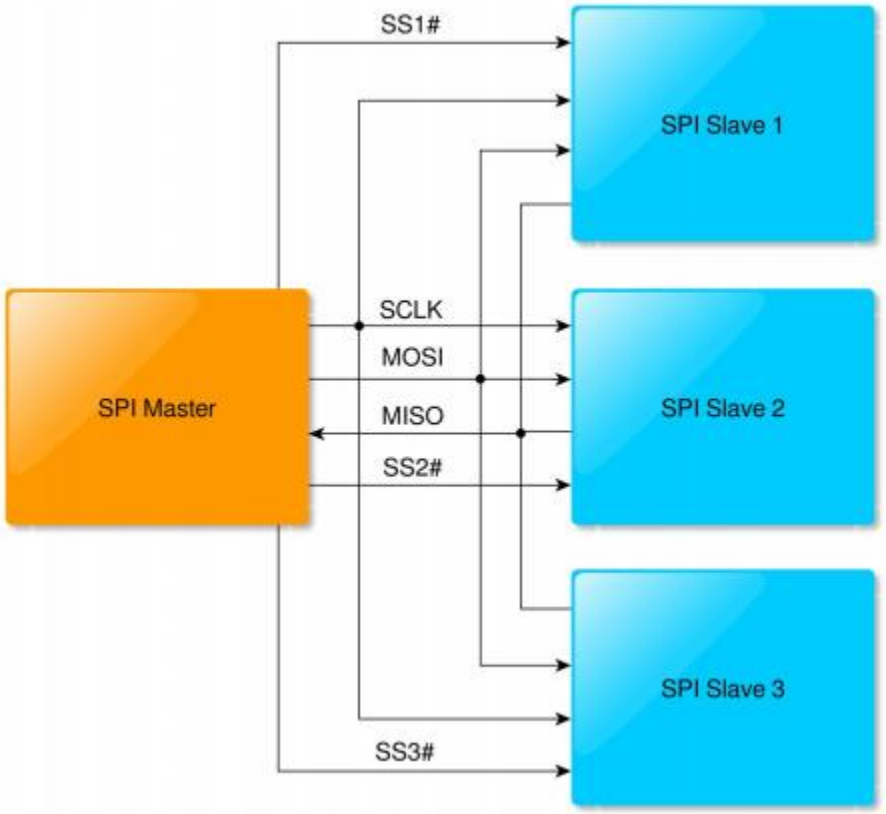
- What is SPI?
- Example SPI Devices
- SPI Modes
- Linux SPI Subsystem
- Linux SPI Drivers
  - Controller Drivers
  - Protocol Drivers
    - Kernel APIs
- Instantiating SPI Devices
- User space tools
- Demo

What is SPI?

- SPI (Serial Peripheral Interface) is a full duplex synchronous serial master/slave bus interface
- De facto standard first developed at Motorola in the 1980s
- A SPI bus consists of a single master device and possibly multiple slave devices
- Typical device interface
  - SCLK – serial clock
  - MISO – master in slave out
  - MOSI – master out slave in
  - CSn / SSn – chip select / slave select
  - IRQ / IRQn – interrupt

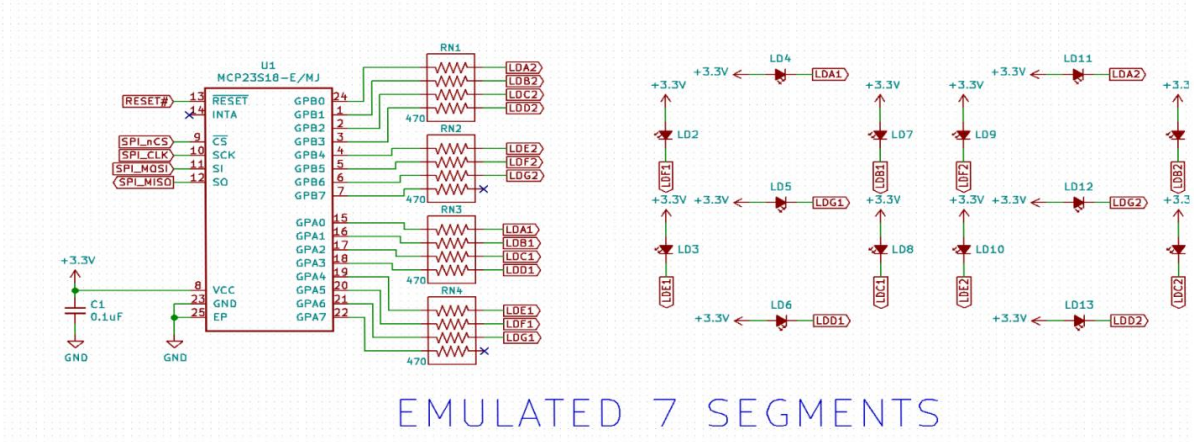


What is SPI?


SS1#
SPI Slave 1
SCLK
MOSI
SPI Master
MISO
SPI Slave 2
SS2#
SPI Slave 3
SS3#

Example SPI devices

- Analog converters (ADC, DAC, CDC)
- Sensors (inertial, temperature, pressure)
- Serial LCD
- Serial Flash
- Touchscreen controllers
- FPGA programming interface

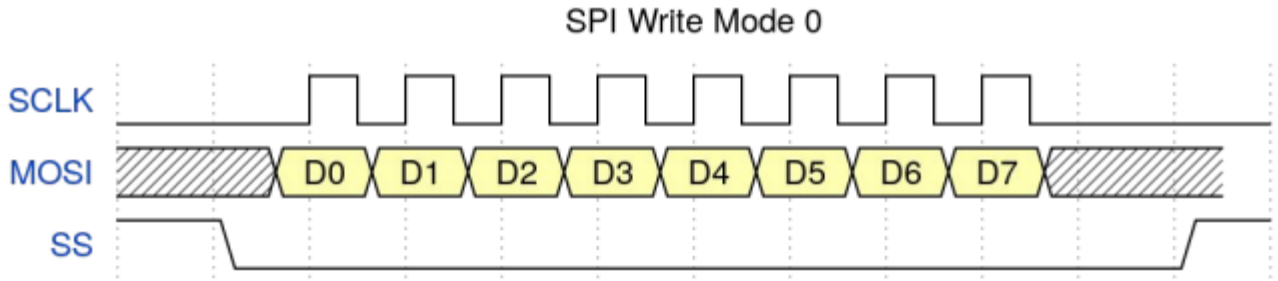


SPI Modes

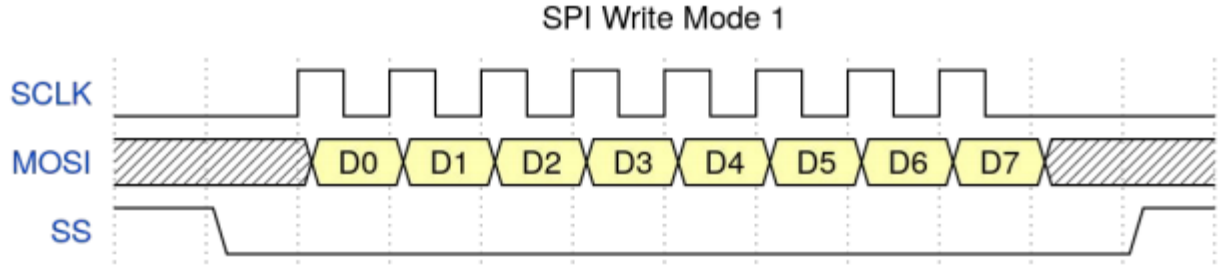
- SPI Mode is typically represented by (CPOL, CPHA) tuple
  - CPOL – clock polarity
    - 0 = clock idles low
    - 1 = clock idles high
  - CPHA – clock phase
    - 0 = data latched on falling clock edge, output on rising
    - 1 = data latched on rising clock edge, output on falling
- Mode (0, 0) and (1, 1) are most commonly used
- Sometimes listed in encoded form 0-3

SPI Modes

SPI Mode Timing – CPOL = 0



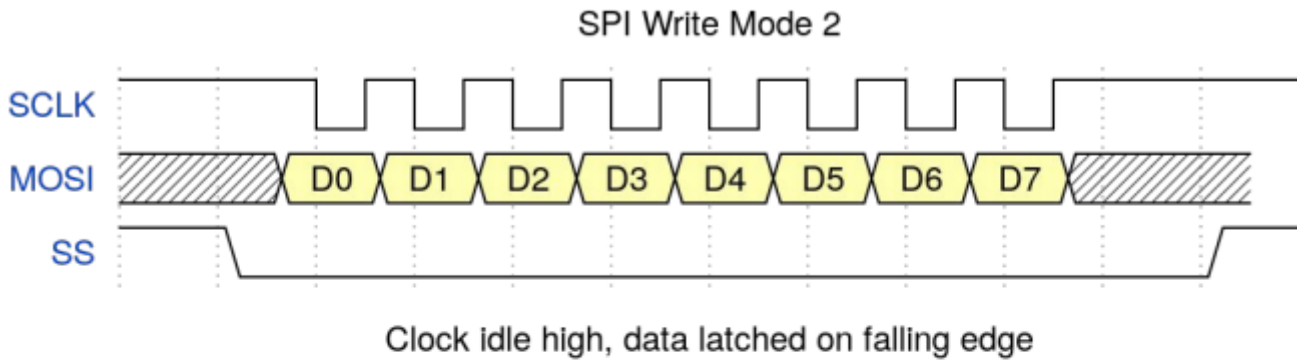
Clock idle low, data latched on rising edge

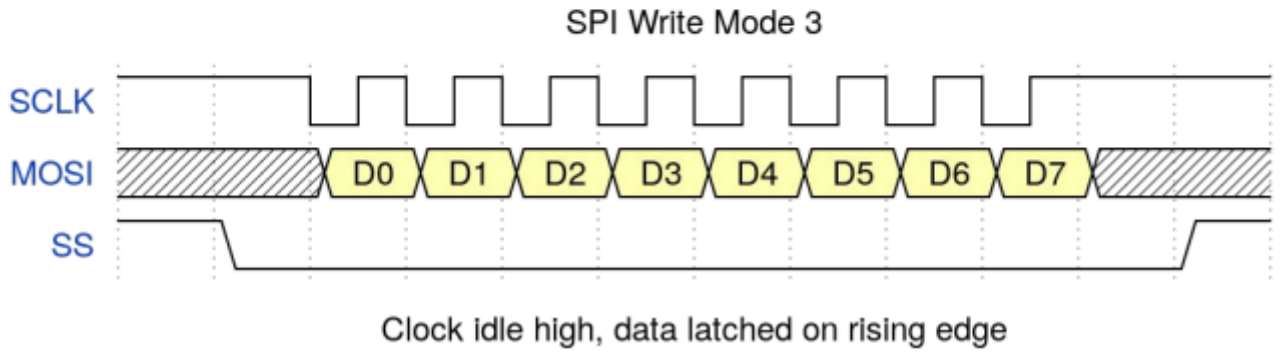


Clock idle low, data latched on falling edge

SPI Modes

SPI Mode Timing – CPOL = 1





Linux SPI Subsystem

First developed in early 2000s (2.6 ERA) based on the work of several key developers in including:

- David Brownell
- Russell King
- Dmitry Pervushin
- Stephen Street
- Mark Underwood
- Andrew Victor
- Vitaly Wool

Linux SPI Subsystem

Past maintainers of the Linux SPI subsystem:

- David Brownell
- Grant Likely

Current maintainer:

- Mark Brown

Linux SPI Subsystem

List: linux-spi; (subscribe / unsubscribe)

Info:

This is the mailing list for the Linux SPI subsystem.

Archives: http://marc.info/?l=linux-spi

Controller Drivers

- Controller drivers are used to abstract and drive transactions on an SPI master.
- The host SPI peripheral registers are accessed by callbacks provided to the SPI core driver. (drivers/spi/spi.c)
- struct spi_controller

Controller Drivers

- Allocate a controller
  - **spi_alloc_master()**
- Set controller fields and methods
  - mode_bits - flags e.g. **SPI_CPOL, SPI_CPHA, SPI_NO_CS, SPI_CS_HIGH, SPI_RX_QUAD, SPI_LOOP**
  - **.setup()** - configure SPI parameters
  - **.cleanup()** - prepare for driver removal
  - **.transfer_one_message()/.transfer_one()** - dispatch one msg/transfer (mutually exclusive)
- Register a controller
  - **spi_register_master()**

Controller Devicetree Binding

The SPI controller node requires the following properties:

- compatible - Name of SPI bus controller following generic names recommended practice.

In master mode, the SPI controller node requires the following additional properties:

- #address-cells - number of cells required to define a chip select address on the SPI bus.
- #size-cells - should be zero.

Optional properties (master mode only):

- cs-gpios - gpios chip select.
- num-cs - total number of chipselects.

So if for example the controller has 2 CS lines, and the cs-gpios property looks like this:
cs-gpios = <&gpio1 0 0>, <0> , <&gpio1 1 0>, <&gpio1 2 0>;

Controller Devicetree Binding

Example:

```
spi1: spi@481a0000 {
        compatible = "ti,omap4-mcspi";
        #address-cells = <1>;
        #size-cells = <0>;
        reg = <0x481a0000 0x400>;
        interrupts = <125>;
        ti,spi-num-cs = <2>;
        ti,hwmods = "spi1";
        dmas = <&edma 42 0
        &edma 43 0
        &edma 44 0
        &edma 45 0>;
        dma-names = "tx0", "rx0", "tx1", "rx1";
        status = "disabled";
};
```

Protocol Drivers

- For each SPI slave you intend on accessing, you have a protocol driver. SPI protocol drivers can be found in many Linux driver subsystems (iio, input, mtd).
- Messages and transfers are used to communicate to slave devices via the SPI core and are directed to the respective controller driver transparently.
- A struct spi_device is passed to the probe and remove functions to pass information about the host.

Protocol Drivers

- Transfers
  - A single operation between master and slave
  - RX and TX buffers pointers are supplied
  - Option chip select behavior and delays

- Messages
  - Atomic sequence of transfers
  - Argument to SPI subsystem read/write APIs

Protocol Drivers

struct spi_device

```
struct spi_device {
        struct device dev;
        struct spi_controller * controller;
        struct spi_controller * master;
        u32 max_speed_hz;
        u8 chip_select;
        u8 bits_per_word;
        u16 mode;
        int irq;
        void * controller_state;
        void * controller_data;
        char modalias;
        int cs_gpio;
        struct spi_statistics statistics;
}
```

Controller side proxy for an SPI slave device. Passed to the probe and remove functions with values based on the host configuration.

Protocol Drivers

```
#define SPI_CPHA 0x01                   /* clock phase */
#define SPI_CPOL 0x02                   /* clock polarity */
#define SPI_MODE_0 (0|0)                /* (original MicroWire) */
#define SPI_MODE_1 (0|SPI_CPHA)
#define SPI_MODE_2 (SPI_CPOL|0)
#define SPI_MODE_3 (SPI_CPOL|SPI_CPHA)
#define SPI_CS_HIGH 0x04                /* chipselect active high? */
#define SPI_LSB_FIRST 0x08              /* per-word bits-on-wire */
#define SPI_3WIRE 0x10                  /* SI/SO signals shared */
#define SPI_LOOP 0x20                   /* loopback mode */
#define SPI_NO_CS 0x40                  /* 1 dev/bus, no chipselect */
#define SPI_READY 0x80                  /* slave pulls low to pause */
#define SPI_TX_DUAL 0x100               /* transmit with 2 wires */
#define SPI_TX_QUAD 0x200               /* transmit with 4 wires */
#define SPI_RX_DUAL 0x400               /* receive with 2 wires */
#define SPI_RX_QUAD 0x800               /* receive with 4 wires */
```

Protocol Drivers
Probe Function

```
static int myspi_probe(struct spi_device *spi)
{
        struct myspi *chip;
        struct myspi_platform_data *pdata, local_pdata;
        ...
}
```

Protocol Drivers
Probe Function

```
static int myspi_probe(struct spi_device *spi)
{
        ...
        match = of_match_device(of_match_ptr(myspi_of_match), &spi->dev);
        if (match) {
                /* parse device tree options */
                pdata = &local_pdata;
                ...
        }
        else {
                /* use platform data */
                pdata = &spi->dev.platform_data;
                if (!pdata)
                        return -ENODEV;
        }
        ...
}
```

Protocol Drivers
Probe Function

```
static int myspi_probe(struct spi_device *spi)
{
        ...
        /* get memory for driver's per-chip state */
        chip = devm_kzalloc(&spi->dev, sizeof *chip, GFP_KERNEL);
        if (!chip)
                return -ENOMEM;

        spi_set_drvdata(spi, chip);
        ...

        return 0;
}
```

Protocol Drivers
OF Device Table

Example:

```
static const struct of_device_id myspi_of_match[] = {
	{
		.compatible = "mycompany,myspi",
		.data = (void *) MYSPI_DATA,
	},
	{ },
};
MODULE_DEVICE_TABLE(of, myspi_of_match);
```

Protocol Drivers

SPI Device Table

Example:

```
static const struct spi_device_id myspi_id_table[] = {
	{ "myspi", MYSPI_TYPE },
	{ },
};
MODULE_DEVICE_TABLE(spi, myspi_id_table);
```

Protocol Drivers

struct spi_driver

```
struct spi_driver {
        const struct spi_device_id * id_table;
        int (* probe) (struct spi_device *spi);
        int (* remove) (struct spi_device *spi);
        void (* shutdown) (struct spi_device *spi);
        struct device_driver driver;
};
```

Protocol Drivers

struct spi_driver

Example:

```
static struct spi_driver myspi_driver = {
        .driver = {
                .name = "myspi_spi",
                .pm = &myspi_pm_ops,
                .of_match_table = of_match_ptr(myspi_of_match),
        },
        .probe = myspi_probe,
        .id_table = myspi_id_table,
};
module_spi_driver(myspi_driver);
```

Protocol Drivers
Kernel APIs

- **spi_async()**
  - asynchronous message request
  - callback executed upon message complete
  - can be issued in any context
- **spi_sync()**
  - synchronous message request
  - may only be issued in a context that can sleep (i.e. not in IRQ context)
  - wrapper around spi_async()
- **spi_write()/spi_read()**
  - helper functions wrapping spi_sync()

Protocol Drivers
Kernel APIs

- **spi_read_flash()**
  - Optimized call for SPI flash commands
  - Supports controllers that translate MMIO accesses into standard SPI flash commands
- **spi_message_init()**
  - Initialize empty message
- **spi_message_add_tail()**
  - Add transfers to the message's transfer list

Instantiating SPI Devices
Slave Node Devicetree Binding

SPI slave nodes must be children of the SPI controller node.

In master mode, one or more slave nodes (up to the number of chip selects) can be present.

Required properties are:
- compatible - Name of SPI device following generic names recommended practice.
- reg - Chip select address of device.
- spi-max-frequency - Maximum SPI clocking speed of device in Hz.

Instantiating SPI Devices
Slave Node Devicetree Binding

All slave nodes can contain the following optional properties:

- spi-cpol - Empty property indicating device requires inverse clock polarity (CPOL) mode.
- spi-cpha - Empty property indicating device requires shifted clock phase (CPHA) mode.
- spi-cs-high - Empty property indicating device requires chip select active high.
- spi-3wire - Empty property indicating device requires 3-wire mode.
- spi-lsb-first - Empty property indicating device requires LSB first mode.
- spi-tx-bus-width - The bus width that is used for MOSI. Defaults to 1 if not present.
- spi-rx-bus-width - The bus width that is used for MISO. Defaults to 1 if not present.
- spi-rx-delay-us - Microsecond delay after a read transfer.
- spi-tx-delay-us - Microsecond delay after a write transfer

Instantiating SPI Devices

Slave Node Devicetree Binding

Example:

```
&spi1 {
        #address-cells = <1>;
        #size-cells = <0>;
        status = "okay";
        pinctrl-names = "default";
        pinctrl-0 = <&spi1_pins>;
        myspi@0 {
                compatible = "mycompany,myspi";
                spi-max-frequency = <2000000>;
                spi-cpha;
                ...
                reg = <0>;
        };
...
};
```

Instantiating SPI Devices

Platform Registration

```
struct spi_board_info {
        char modalias;
        const void * platform_data;
        const struct property_entry * properties;
        void * controller_data;
        int irq;
        u32 max_speed_hz;
        u16 bus_num;
        u16 chip_select;
        u16 mode;
};
```

Instantiating SPI Devices
Platform Registration

Example:

```
static struct spi_board_info myspi_board_info[] = {
 {
        .modalias = "myspi",
        .platform_data = &myspi_info,
        .irq = MYIRQ,
         .max_speed_hz = 2000000,
        .chip_select = 2,
        …..
        },
};
```

User space tools
spidev

What does spidev do?

- Passes data between user space and SPI controller
- Collects buffers for TX/RX from user space application
- Hands off buffers to SPI controller driver
- Returns to user space when transfer is complete

User space tools
spidev

When should spidev be used?

- Prototyping in an environment that's not crash-prone; stray pointers in user space won't normally bring down any Linux system.
- Developing simple protocols used to talk to microcontrollers acting as SPI slaves, which you may need to change quite often.

https://www.kernel.org/doc/Documentation/spi/spidev

User space tools
spidev

When should spidev NOT be used?

- Of course there are drivers that can never be written in user space, because they need to access kernel interfaces (such as IRQ handlers or other layers of the driver stack) that are not accessible to user space.

https://www.kernel.org/doc/Documentation/spi/spidev

User space tools

spidev

SPI devices have a limited user space API, supporting basic half-duplex **read()** and **write()** access to SPI slave devices. Using **ioctl()** requests, full duplex transfers and device I/O configuration are also available.

Required header files:

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <linux/types.h>
#include <linux/spi/spidev.h>
```

User space tools
spidev

The sysfs node for the SPI device will include a child device node with a “dev” attribute that will be understood by udev or mdev.

For a SPI device with chip select C on bus B, you should see:

- **/dev/spidevB.C** - character special device, major number 153 with a dynamically chosen minor device number.
- **/sys/devices/…/spiB.C** - SPI device node will be a child of its SPI master controller.
- **/sys/class/spidev/spidevB.C** - created when the “spidev” driver binds to that device.

User space tools

spidev

Normal **open()** and **close()** operations on /dev/spidevB.D files work as you would expect.

Standard **read()** and **write()** operations are obviously only half-duplex, and the chipselect is deactivated between those operations.

Full-duplex access, and composite operation without chipselect de-activation, is available using the **SPI_IOC_MESSAGE(N)** request.

User space tools
spidev ioctl

Several ioctl() requests let your driver read or override the device's current settings for data transfer parameters:

SPI_IOC_RD_MODE, SPI_IOC_WR_MODE

Pass a pointer to a byte which will return (RD) or assign (WR) the SPI transfer mode. Use the constants **SPI_MODE_0..SPI_MODE_3**; or if you prefer you can combine **SPI_CPOL** (clock polarity, idle high iff this is set) or **SPI_CPHA** (clock phase, sample on trailing edge iff this is set) flags. Note that this request is limited to SPI mode flags that fit in a single byte.

SPI_IOC_RD_MODE32, SPI_IOC_WR_MODE32

Pass a pointer to a uin32_t which will return (RD) or assign (WR) the full SPI transfer mode, not limited to the bits that fit in one byte.

User space tools
spidev ioctl

SPI_IOC_RD_LSB_FIRST, SPI_IOC_WR_LSB_FIRST

Pass a pointer to a byte which will return (RD) or assign (WR) the bit justification used to transfer SPI words. Zero indicates MSB-first; other values indicate the less common LSB-first encoding. In both cases the specified value is right-justified in each word, so that unused (TX) or undefined (RX) bits are in the MSBs.

SPI_IOC_RD_BITS_PER_WORD, SPI_IOC_WR_BITS_PER_WORD

Pass a pointer to a byte which will return (RD) or assign (WR) the number of bits in each SPI transfer word. The value zero signifies eight bits.

SPI_IOC_RD_MAX_SPEED_HZ, SPI_IOC_WR_MAX_SPEED_HZ

Pass a pointer to a u32 which will return (RD) or assign (WR) the maximum SPI transfer speed, in Hz. The controller can’t necessarily assign that specific clock speed.

User space tools
spidev

```
__u8 miso[MAX_LENGTH];
__u8 mosi[MAX_LENGTH];

struct spi_ioc_transfer tr = {
        .tx_buf = (unsigned long)mosi,
        .rx_buf = (unsigned long)miso,
        .delay_usecs = 1,
        .len = 1,
};
…
fd = open(device_name, O_RDWR);
…
ret = ioctl(fd, SPI_IOC_MESSAGE(1), &tr);
```

https://github.com/mwelling/spi-test/

Demo

Questions?

Embedded Linux Conference

North America